The Need for Inner-Procedural Refinements

Lutz Prechelt (prechelt@ira.uka.de)
Institut für Programmstrukturen und Datenorganisation
Universität Karlsruhe
D–7500 Karlsruhe, Germany

November 23, 1992

Abstract

Today one of the most widely accepted paradigms of software engineering is the Top-Down method. Based on the hypothesises that (a) Top-Down is a good way to go and (b) programmers are lazy by their very nature, I show here that with most modern imperative programming languages, we stop the Top-Down process at least one level too early. What languages need is the concept of refinements within procedures, so the Top-Down development process can continue arbitrarily deep inside each procedure. This leads to better-structured code with improved understandability. Existing languages can be extended to support the notion of inner-procedural refinements. It is described how this was done with a fast preprocessor for C and C++ and what the experiences with it are.

Key words: refinements, C, programming language extension, Top-Down method

The Top-Down design process

Let us take a quick look on why Top-Down is a good method. We will see that the ultimate reason lies in the human nature.

Today we build a huge software system by first decomposing it into subsystems that interact through a (hopefully) well designed protocol. Each of these subsystems is then divided into a hierarchical collection of modules and/or classes. Each of these modules or classes consists (among other things) of several procedures or other kinds of subprograms, some of which represent the interface of the module or class, and some of which are merely there to master the complexity of the module or class. (In this respect object-oriented programming is just a particular form of the Top-Down method.)

The underlying paradigm of Top-Down design and development is 'divide and conquer'. From a cognitive point of view this means to separate the task of answering the question *what must I do* from the task of answering the question *how should I do it*, then solving these two problems one after the other and applying the same method to each of the emerging subproblems recursively.

This is a relatively disciplined way to use ones mind, and thus should not at all be expected to be comfortable for a human being. So, why is it that widely used? The answer is: because Top-Down can be used in a way that it means *always think as late as possible*, and this is of course highly attractive to lazy minds.

Where we stop today

The common imperative programming languages that explicitly support Top-Down design and programming like ADA, Modula–2 and C++ have two main concepts for Top-Down decomposition: The module or class and the procedure or function (or the like).

These languages were designed from a software engineering perspective; and in fact everything seems to be alright from a pure software engineering point of view: data abstraction, algorithmic abstraction and data encapsulation are supported; it is possible to hide implementation decisions concerning the data structures and the algorithms used. This leads to a certain degree of independence of the modules; big software systems become manageable and everything looks fine.

Why we should go further

But remember the second main assumption: programmers are lazy.

As we already have seen, Top-Down suits this fine: as long as you go from top down, you constantly defer parts of the heavy task of thinking. It is easy to see that it would be nice for the programmer to continue with this mode of operation until the level of single statements or small statement groups and simple expressions is reached.

Today, what do you do, when your procedure gets complicated? You can break it up into several procedures. But will you really always do that? No. The work that is necessary due to the syntactic overhead of this action will often hinder you. Consider the following examples:

Example 1: A procedure is 50 lines long, no tricky code, but control structures (ifs, loops etc.) have to be nested 5 levels deep.

Example 2: A procedure performs its task in 5 stages. Nothing complicated, but more than 100 lines long.

Example 3: A procedure consists of an if–then–else–if with 8 parts. The boolean expressions each have about 10 factors.

In probably none of these cases an average programmer would see the need for further Top-Down decomposition, but the resulting code will be difficult to read. Sometimes heavy commenting will be used or temporary variables, but this is neither self evident, because these are steps that take some effort to perform instead of saving effort, nor is it sufficient, because the layout of the code will still be complicated and will not make the underlying structure easy for people to find.

What could solve the problem, is a way to replace the nested parts or stages or subexpressions with symbolic names, provided that the effort necessary to introduce these names is

close to zero. The decomposition with such symbolic names would recursively be applied until the remaining pieces of code have a size of about, say, 10 statements or 2 factors in an expression. Such names must be local to the procedure to avoid name conflicts. The pieces of code can thus be called inner-procedural refinements or just *refinements* and the resulting style of programming can be viewed as a low-level application of the principle of stepwise refinement as described by Wirth [1].

How to do it

What properties must such inner-procedural refinements have in order to be useful ?

First, one must be able to use a refinement *before* its definition, so that one can easily defer thinking about what exactly he/she means with it but nevertheless write down the program without having to hop backwards textually. Second, no lengthy keywords or the like must be necessary to declare or use a refinement. Third, since this part of the Top-Down process is local to the procedure, the refinements have to be local to the procedure, too.

As far as I know, the only imperative programming language that incorporates these considerations in its design is ELAN [2].

The C–Refine Preprocessor

Existing languages can be extended to support refinements. The following syntax for the extension of C [3] satisfies the above needs. It is implemented in a preprocessor called C–Refine.

Example (just to show the syntax):

```
void an_example_for_refinements (a, b, x, y)
{
  if (‘some test)          /* The overall structure of */
    ‘my first refinement;  /* the function can be seen */
  else                     /* at once */
    ‘my second refinement;

‘some test:
   a < b && ‘some subtest

‘some subtest:
   a != 0 || b != 0

‘my first refinement:
   ‘my first subrefinement;
   statement2;

‘my first subrefinement:
   while (x) {
```

```
    /* some statements here */
    if (y)
      leave `my first refinement;  /* structured "goto" */
    /* ...lots of what-to-do-else here */
  }
  /* ...more lots of what-to-do-else here */

`my second refinement:
  /* and so on... */

}
```

The syntax is based on the use of a special character (the back quote) which is used to mark the names of refinements and on the formatting convention that a refinement declaration must begin in column 0.

The `leave` construct generates a `goto` to the end of the body of the refinement given with it, so multiple levels of loops etc. can be exited from at once.

Implementation and Speed Considerations

The C–Refine preprocessor is bootstrapped and knows almost nothing about the syntax and semantics of C. This has two effects: the same preprocessor can be used for all languages that are similar to C (such as C++) and it is very fast.
The underlying algorithm can be described roughly as follows: read the input file and look for opening and closing braces. Any such block on the outermost level is considered a function body. Copy everything that is outside any function body from input to output and buffer anything that is inside any function body. While reading a function body store the positions of all refinement bodys along with the names of the refinements. Whenever a function body is complete write the buffered lines to the output, replacing every refinement call with a copy of the corresponding refinement body (recursively) and replacing every *leave*construct with a goto statement.

The actual algorithm is only a bit more complicated than this outline. The significant point is that no parsing of the language is needed, a simple scanner suffices.

A test with a 14000 line input file (concatenated from 35 real modules) took 2.5 seconds on a Sun 4/20, 11 seconds on a Sun 3/160 and 14 seconds on a Sun 3/50. This is multiple times faster than the C compiler on the same machines executes.

Since the preprocessor is so simple, it can easily be ported on any Unix machine. I have already tried it on Sun-OS 3.5, 4.0.3 and 4.1, Ultrix 2.1 (DEC), a System V machine (PCS) and even MS-DOS.

Experiences with C–Refine

Learning Effort and Ease of Use

Since the syntax is very intuitive, almost all C programmers can (without any preparation) read a C–Refine program after looking at it for a minute.

The experience shows that it is not always easy to convince a programmer, that using C–Refine is advantageous. But once he or she is willing to try, the use is learnt in less than an hour and all people who have ever tried C–Refine keep using it.

Real Applications

The C–Refine preprocessor has been used in at least two commercial projects with C and is in heavy use for scientific programming in C++ [4]. We also apply it to MPL, the C-derived language for the MasPar MP-1 massively parallel SIMD machine [5].

No technical problems have been encountered.

Observations

No empirical study with controlled conditions in order to quantify the usefulness of C–Refine for the coding process has been conducted so far. Our observations in daily work suggest the following:

1. It is much easier to read or modify other people's programs if they are written in C–Refine as opposed to C. The analog is true for C++.
2. It is also much easier to read the own programs, be it after several months or while the coding process is still going on.
3. Coding tends to be faster in C–Refine as opposed to C as soon as the procedures have more than a certain degree of complexity. Coding may take a little longer for very easy procedures. In either case, the observations (1) and (2) apply.
4. Sometimes programming errors are avoided by the following phenomenon: The programmer wants to skip some subtask intermediately when coding ("I'll do that later"). When using normal C, the corresponding statement(s) may simply be forgotten in the end, but when using C–Refine, it is common that one ends up with an undeclared refinement instead, since it is normal C–Refine style to write down a name for something to be coded later.
5. Complex expressions (especially boolean expressions) contain less errors, because they are systematically split into simpler ones and end up in an easily verifyable self documenting form.

Why this really works

Now why should we think that just this method of solving the problem actually works while most others do not ?

The reason is that with the above syntax the effort (measured in number of keystrokes) to write a program down is increased only very little, but the effort to read it (measured in number of seconds to see what its structure is and to understand what it does) is reduced a lot. This is a fact that the programmer quickly gets in touch with, because a program

must be written only once, but will be read many many times even if we consider only the day when it is actually programmed.

In this respect the basic idea behind C–Refine differs substantially from that of WEB [6]: The goal is not to get a nice documentation for the program after some processing steps, but instead to maximize the readability of the source code itself. This type of readability can be exploited already in the ongoing coding process.

After this observation has been made by a certain programmer, he or she will use refinements heavily and the advantageous effects described above emerge.

This reduction in the cognitive effort for reading a program, be it in the initial programming process or in all the repititions of re-understanding the own or understanding an unknown program, is probably the main reason why programming with inner-procedural refinements has proved to be a good idea.

References

[1] Nikolaus Wirth. Program development by stepwise refinement. *Communications of the ACM*, 14(4), April 1971.

[2] Günther Hommel, Joachim Jäckel, Stefan Jähnichen, Karl Kleine, Wilfried Koch, and Kees Koster. *ELAN Sprachbeschreibung.* Akademische Verlagsgesellschaft, Wiesbaden, Deutschland, 1979.

[3] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language.* Prentice Hall, 1977.

[4] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual.* Addison Wesley, Reading, Mass., 1990.

[5] MasPar Computers, Sunnyvale, Calif. *MPL Language Reference Manual.*

[6] Donald Ervin Knuth. The WEB system of structured documentation. Technical report, Stanford University, Department of Computer Science, Stanford, Calif., 1983.